Chapter 2: Objects and Messages

Return to [Table of Contents] [Main Page] [Previous Chapter] [Next Chapter]

Sending Messages to Objects

As mentioned before, all Smalltalk processing is accomplished by sending *messages* to *objects*. An initial problem solving approach in Smalltalk is to try to *reuse* the existing objects and messages. The Smalltalk programmer works to develop an object-message sequence that will provide the desired solution. (See also "Object-Oriented Problem Solving Approach".)

Objects are instances of a particular *class*. The messages that an object can respond to is defined in the protocol of its *class*. How messages are executed or implemented is defined in the class *methods*. Methods give the implementation details for the messages and represent a class behavior.

Before going any further, let's take a look on how objects and messages are named.

Object and Message Naming

In this tutorial, all *class* names begin with a capital letter such as "**Student**". *Message* names begin with a lower case letter and can have any combination of letters and numbers without embedded blank spaces. When a message name contains more than one word, the extra words start with a capital letter. A valid message name would be "**aMessage**" or "**myAddress**."

A message such as "give me your name" could be "**giveMeYourName**". This however, is too difficult to work with. It would be easier to shorten it to just "**name**".

A message that passes information or *parameters*, such as "set the name to this name", is handled similarly, except that the method name must end with a colon (:). So, The message name becomes "**name:**". These types of messages are called *keyword* messages. Messages that have no information to pass are *unary* messages.

A message can contain multiple arguments. There has to be a keyword for every argument in the message. For example, it is possible to set both the name and the address of a student with one message. This requires a message name with two keywords such as "**name:address:**".

There are three types of messages: *unary*, *binary* and *keyword* .

Unary Messages

An *Unary message* is similar to a single-parameter function call. This type of message consists of a message name and an operand. The objects are put before the message name. The following are examples of unary message:

```
x sin                   "return the result of sin(x)"

Date tomorrow           "answer a new instance of class Date"

5 factorial             "return the factorial of 5"

`hi' outputToPrinter    "Send the string `hi' to the printer"
```

In these examples `x', `Date', `5', and `hi' are the objects and `sin', `tomorror', `factorial', and `outputToPrinter' are the message names.

Binary Messages

Binary messages are used to specify arithmetic, comparison, and logical operations. A binary message can be either one or two characters long and can contain any combination of the following special characters:

```
+ / \ * ~ < > = @ % | & ? ! ,
```

The following are examples of Smalltalk expressions using binary messages:

```
a + b           "returns the result of sum a and b"

a | b           "returns the result of "ORing" a with b"

a >= b          "compares to see if a is greater than or
                 equal to b and returns either true or false"
```

The first example can also read as "the message `+' is sent to the object *`a'* with a parameter *`b'*."

Keyword Messages

A *Keyword message* is equalvalent to a procedure call with two or more parameters. Look at the following example, the name of the object to which the message is sent is written first, then the name of the message (or method name), and follow by the parameter to be passed.

```
AnObject aMessage: parameter
```

The colon is a required part of the message name.

When there is more than one parameter, a message name must appear for each parameter.

For example:

```
AnObject aName1: parameter1 aName2: parameter2
```

Here:

- **AnObject** is the receiving object
- **aName1:** is the first part of the message name
- **parameter1** is passed to **aName1:**
- **aName2:** is the second part of the message name
- **parameter2** is passed to **aName2:**

Examples of keyword messages:

```
Array new: 20              "The message new: is sent to the object
                            Array with parameter 20"

TheDate month: currentMonth year: currentYear
                           "Set the private data for object TheDate
                            to month = currentMonth and year = currentYear"

Student name: `Steve' address: `Raleigh,
                           "Set the appropriate variables in the
                            Student object according to the parameters"
```

Message Format

In general, a Smalltalk expression consists of the name of the object that receives the message, followed by the message name.

For example:

```
AnObject aMessage
```

For messages that need to pass arguments to the receiving object, such as a keyword message, the receving object's name is followed by the message name and its argument. The arguments are separated by a blank space.

```
AnObject aNumber: 1 aName: `John'
```

Object Identifiers

When an object is created, Smalltalk assigns an *identifier* to it. Each object has its own unique identifier that has meaning only to the Smalltalk system.

A variable in Smalltalk contains the identifier of the object, not the object itself. In other words, a variable *points* to an object. When the receiver of a message is a variable, Smalltalk uses the contents of the variable to identify the object. For example, assume that there is a *global variable* called **MyStudent** that points to a **Student** object. The global variable, **MyStudent** can now be the receiver of the following messages:

```
MyStudent name: `Steve'

MyStudent name
```

Creating New Instances

To create a new instance for a class, the **new** message is sent to the class.

For example:

Student new

Every class automatically supports the **new** message. The **new** message then returns a pointer to the new instance.

In the following example, the global variable **MyStudent** points to a new **Student** object:

```
MyStudent := Customer new
```

In Smalltalk, the **:=** expression is the syntax for the assignment function. The variable on the left points to the result of the expression on the right.

Once the statement above has been executed, messages can be sent to **MyStudent**, which points to an instantiation of a **Student** object.

For example:

```
MyStudent name: `Steve'

MyStudent name
```

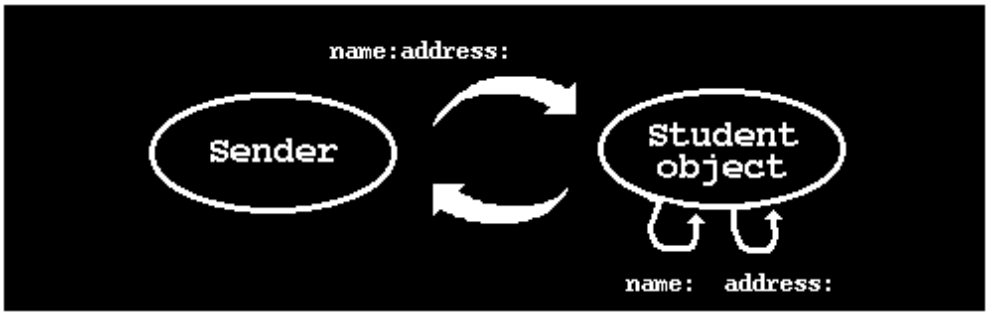
Concept of self

Consider the following statement:

```
MyStudent name: `Steve' address: `Raleigh, NC'.
```

The statement sends the message **name:address:** to the **Student** object pointed to by the variable **MyStudent**. Rather than duplicating the code that already exists in the **name:** and **address:** methods, the method **name:address:** is implemented to have **name:** and **address:** perform the task. Since **name:** and **address:** are methods of the same object as **name:address:**, there must be a way for a method to refer to the object in which it exists.

That is the purpose of self. A method can access other methods in its own object by specifying *self* as the receiver of the message.



Let's look at the code for all three of these methods:

```
name: aName
      name := aName

address: anAddress
      address := anAddress

name: aName address: anAddress
      self name: aName.
      self address: anAddress
```

The use of *self* causes Smalltalk to send the **name** and **address** messages to the current instance of **Student**, the same instance that received the **name:address:** message.

In the example, the methods **name:** and **address:** defined their temporary variables with the same identities that are found in **name:address:**. There is no relationship between these names and they do not have to be the same. The scope of a method's temporary variables are local only to that method. For example, the **name:** mehtod could also look like this:

```
name: studentName
    name := studentName
```

The **name:address:** method could choose to directly set the variables instead of using the lower-level methods of **name:** and **address:**. This would appear as follows:

```
name: aName address: anAddress
    name := aName
    address := anAddress
```

It is better, however, to centralize the setting of a variable inside a single method. The same is true for getting the contents of a variable.

Order of Message Execution

The rules that Smalltalk use when deciding the order of exercuting messages can be summarize as the following:

1. Smalltalk executes messages from left to right.
2. The result of a message replaces that message in the statement.
3. Smalltalk executes all expression that appear inside a pair of parenthesis first, with the left-most inner pair of nested parenthesis.
4. Inside an expression, unary messages are executed first, followed by binary messages, followed by keyword messsage, in a left-to-right direction.
5. Smalltalk executes all binary messages from left to right, regardless of what operations they perform. This means there is no special order of execution for arithmetic operations.
6. An expression can include a variable name. Smalltalk replaces the variable name with the object to which it points.

Summary

We have discussed the following in this chapter:

- All Smalltalk processing involves sending **messages** to **objects**. **Class methods** define the behavior of the class messages.

- Class names begin with capital letters and message names begin with lowercase letters. The three types of messages are: **unary, binary,** and **keyword**.

- A general **Smalltalk expression** consist of the name of the object that receives the message followed by the message name and its arguments.

- An object can send a message to itself by specifying the name **self** as the receiving object.

- Smalltalk has a series of execution-time rules for evaluating a Smalltalk statement. There are four important rules for evaluating code:

1. All code is evaulated left to right.
2. Expressions within parentheses are evaluated first, starting with the left-most inner parenthesis.
3. The result of a message takes the place of that message in an expression.
4. **Unary messages** are executed first, followed by **binary messages** , followed by **keyword messages**.

Return to [Top of the page]

Smalltalk Tutorial Contents

Go to Chapter 3: Smalltalk Statements

Return to Chapter 1: Introduction to IBM Smalltalk Programming

Return to Main Page

Smalltalk Message Rules

The basic rules for messages/methods in SmalltalkLanguage are:

- **Unary messages:** Take no arguments.

```
Dictionary new.
12 factorial.
```

- **Binary messages:** Take exactly one argument and use one or more non-alphabetic symbols for the selector.

```
12 + 7.
'BIG' > 'little'
```

- **Keyword messages:** Take one or more arguments and use words followed by colon before each argument.

```
5 between: 1 and: 10.
Transcript show: aString.
12 > 7 ifTrue: ['yes'] ifFalse: ['no'].
```

- **Precedence rules:** Left to right, Unary then binary then keyword.

```
 3 squared + 4 squared between: 1 and: 1000.
 Executes 3 squared then 4 squared then 9 + 16 then
25 between: 1 and: 1000.
```

- **Method signatures:** Methods are identified by the symbol which is their selector:

```
 #new, #factorial, #+, #>, #between:and:, #show:,
#ifTrue:ifFalse:
```

Can anyone explain how the messaging notation works when combined with the ; cascading operator? I.e. I have something like:

```
  7 + 3 squared; printOn: aStream
```

What's printed now? 3 or (7 + 3 squared)? -- StephanHouben (trying to write a YACC grammar of Smalltalk, just for the heck of it).

The meaning of semi-colon is, "Send the next message to the same object that received the previous message". A simpler example would be:

```
 aStream
    nextPutAll: 'line 1';
    cr;
    nextPutAll: 'line 2'.
```

So in this case, **cr** is sent to the same object that received the first **nextPutAll:**, namely aStream. Same goes for the last **nextPutAll:**.

Your example is a bit trickier, though, because of the precedence rules:

```
 7 + 3 squared; printOn: aStream
```

```
 3 squared gets evaluated first. Then 7 + 9, then 7
printOn: aStream.
```

Here's a couple more examples:

```
 7 + 4 squared; factorial; yourself.
```

```
 4 squared, then 7 + 16, then 7 factorial then 7
yourself -> 7.


 7 + 2 squared factorial; yourself.


 2 squared, then 4 factorial then 7 + 24, then 7
yourself -> 7.
```

I find it quite easier "parse" the above examples like this (it avoids english :)

```
7
 + 4 squared;
 factorial;
 yourself.


7
 + 2 squared factorial;
 yourself.
```

-- EtoffiPerson

Q: Can anyone explain how Smalltalk implements shortcut evaluation of logical operators? E.g. is this possible?

```
    ((x != 0) && ((n / x) > 10)) ifTrue: [ .... ].
```

Apologies if the operators are not correct -- I'm interested in the semantics, rather than the precise syntax. -- anon

A: That is a form of deferred evaluation, which in Smalltalk is always done with blocks. The example becomes:

```
    (x ~= 0 and: [n / x > 10]) ifTrue: [ .... ].
```

The first part, x ~= 0, returns true or false which are normal objects. When true is sent #and:, it evaluates the second argument (the block [n / x > 10]) with #value and returns that. When false is sent #and:, it just returns false without evaluating the second argument.

Thus what is done in CeePlusPlus with special language magic is done in Smalltalk with the general mechanisms of blocks and polymorphism. One consequence is that you can write your own short-cut methods which behave like the built-in ones. In C++, you can overload operator&&() but you cannot duplicate the short-cut arguments.

Previous edit of this page was 2005 -- in 2012, C++ finally has deferred evaluation with lambdas, so you could write this as:

```
    Iftrue(And(x != 0, [=]{ return (n / x) > 10 }),
[=]{ ... })
```

*What? I didn't say the future was *pretty**

CategorySmalltalk

Last edit January 25, 2012

Download smalltalk (PDF)

Classes and methods

Example

Classes and methods are usually defined in the Smalltalk IDE.

Classes

A class definition looks something like this in the browser:

```
XMLTokenizer subclass: #XMLParser
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'XML-Parser'
```

This is actually the message the browser will send for you to create a new class in the system. (In this case it's `#subclass:instanceVariableNames:classVariableNames:poolDictionaries:category:`, but there are others that also make new classes).

The first line shows which class you are subclassing (in this case it's XMLTokenizer) and the name the new subclass will have (#XMLParser).

The next three lines are used to define the variables the class and it's instances will have.

Methods

Methods look like this in the browser:

```
aKeywordMethodWith: firstArgument and: secondArgument
    "Do something with an argument and return the result."

    ^firstArgument doSomethingWith: secondArgument
```

The `^` (caret) is the return operator.

```
** anInteger
    "Raise me to anInteger"
    | temp1 temp2 |

    temp1 := 1.
    temp2 := 1.
    1 to: anInteger do: [ :i | temp1 := temp1 * self + temp2 - i ].
    ^temp1
```

this is not the right way to do exponentiation, but it shows a *binary message* definition (they're defined like any other message) and some *method temporary variables* (or method temporaries, *temp1* and *temp2*) plus a block argument (*i*).

Download smalltalk (PDF)

Literals and comments

Example

Coments

```
"Comments are enclosed in double quotes. BEWARE: This is NOT a string!"
"They can span
 multiple lines."
```

Strings

```
'Strings are enclosed in sigle quotes.'
'Single quotes are escaped with a single quote, like this: ''.'
''''  "<--This string contains one single quote"

'Strings too can span
multiple lines'

''   "<--An empty string."
```

Symbols

```
#thiIsaSymbol  "Symbols are interned strings, used for method and variable names,
                and as values with fast equality checking."
#'hello world' "A symbol with a space in it"
#''            "An empty symbol, not very useful"
#+
#1             "Not the integer 1"
```

Characters

```
$a     "Characters are not strings. They are preceded by a $."
$A     "An uppercase character"
$      "The spacecharacter!"
$→     "An unicode character"
$1     "Not to be confused with the number 1"
```

Numbers

Numbers come in all varieties:

Integers:

- Decimal

  10

  -1

  0

  1000000000000000000000000000000000000000000000

- Hexadecimal

  16rAB1F

  16r0

  -16rFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

- ScaledDecimal

  17s0

  3.14159265s8

- Other

  8r7731 "octal"

  2r1001 "binary"

  10r99987 "decimal again!"

Floating point

```
3.14 1.2e3     "2 floating-point numbers"
```

Fractions

Fractions are not the same as floating-point numbers, they're exact numbers (no rounding error).

```
4/3             "The fraction 4/3"
355/113         "A rational approximation to pi"
```

Arrays

```
#(#abc 123)    "A literal array with the symbol #abc and the number 123"
```

Byte Arrays

```
#[1 2 3 4]       "separators are blank"
#[]              "empty ByteArray"
#[0 0 0 0 255]   "length is arbitrary"
```

Dynamic arrays

Dynamic arrays are built from expressions. Each expression inside the braces evaluates to a different value in the constructed array.

```
{self foo. 3 + 2. i * 3}   "A dynamic array built from 3 expressions"
```

Blocks

```
[ :p | p asString ] "A code block with a parameter p.
                     Blocks are the same as lambdas in other languages"
```

Some notes:

Note that literal arrays use any kind and number of blanks as separators

```
#(256 16rAB1F 3.14s2 2r1001 $A #this)
"is the same as:"
#(256
  16rAB1F
  3.14s2
  2r1001
  $A #this)
```

Note also that you can compose literals

```
#[255 16rFF 8r377 2r11111111]       (four times 255)

#(#[1 2 3] #('string' #symbol))     (arrays of arrays)
```

There is some "tolerance" to relaxed notation

```
#(symbol) = #(#symbol)              (missing # => symbol)

#('string' ($a 'a'))                (missing # => array)
```

But not here:

```
#([1 2 3]) ~= #(#[1 2 3])           (missing # => misinterpreted)
```

However

```
#(true nil false)                   (pseudo variables ok)
```

But not here!

```
#(self) = #(#self)                  (missing # => symbol)
```

As you can see there are a couple of inconsistencies:

- While pseudo variables `true`, `false` and `nil` are accepted as literals inside arrays, the pseudo variables `self` and `super` are interpreted as **Symbols** (using the more general rule for unqualified strings.)
- While it is not mandatory to write `#(` for starting a nested array in an array and the parenthesis suffices, it is mandatory to write `#[` for starting a nested `ByteArray`.

Some of this was taken from:

http://stackoverflow.com/a/37823203/4309858

Download smalltalk (PDF)

Loops in Smalltalk

Example

For this example, an `Ordered Collection` will be used to show the different messages that can be sent to an `OrderedCollection` object to loop over the elements.
The code below will instantiate an empty `OrderedCollection` using the message `new` and then populate it with 4 numbers using the message `add:`

```
anOrderedCollection := OrderedCollection new.
anOrderedCollection add: 1; add: 2; add: 3; add: 4.
```

All of these messages will take a block as a parameter that will be evaluated for each of the elements inside the collection.

1. `do:`
   This is the basic enumeration message. For example, if we want to print each element in the collection we can achieve that as such:

   ```
   anOrderedCollection do:[:each | Transcript show: each]. "Prints --> 1234"
   ```

   Each of the elements inside the collection will be defined as the user wishes using this syntax: `:each` This `do:` loop will print every element in the collection to the `Transcript` window.

2. `collect:`
   The `collect:` message allows you to do something for each item in the collection and puts the result of your action in a new collection
   For example, if we wanted to multiply each element in our collection by 2 and add it to a new collection we can use the `collect:` message as such:

   ```
   evenCollection := anOrderedCollection collect:[:each | each*2]. "#(2 4 6 8)"
   ```

3. `select:`
   The `select:` message allows you to create a sub-collection where items from the original collection are selected based on some condition being true for them. For example, if we wanted to create a new collection of odd numbers from our collection, we can use the `select:` message as such:

   ```
   oddCollection := anOrderedCollection select:[:each | each odd].
   ```

   Since `each odd` returns a Boolean, only the elements that make the Boolean return true will be added to `oddCollection` which will have `#(1 3)`.

4. `reject:`

   This message works opposite to `select:` and rejects any elements that make the Boolean return `true`. Or, in other words it will select any elements that make the Boolean return `false`. For example if we wanted to build the same `oddCollection` like the previous example. We can use `reject:` as such:

```
oddCollection := anOrderedCollection reject:[:each | each even].
```

   `oddCollection` will again have `#(1 3)` as its elements.

These are the four basic enumeration techniques in Smalltalk. However, feel free to browse the `Collections` class for more messages that may be implemented.

Download smalltalk (PDF)

Message sending

Example

In Smalltalk almost everything you do is *sending messages* to objects (referred as calling methods in other languages). There are three types of messages:

Unary messages:

```
#(1 2 3) size
"This sends the #size message to the #(1 2 3) array.
 #size is a unary message, because it takes no arguments."
```

Binary messages:

```
1 + 2
"This sends the #+ message and 2 as an argument to the object 1.
 #+ is a binary message because it takes one argument (2)
 and it's composed of one or two symbol characters"
```

Keyword messages:

```
'Smalltalk’ allButFirst: 5.
"This sends #allButFirst: with argument 5 to the string 'Smalltalk',
 resulting in the new string 'talk'"

3 to: 10 by: 2.
"This one sends the single message #to:by:, which takes two parameters (10 and 2)
 to the number 3.
 The result is a collection with 3, 5, 7, and 9."
```

Multiple messages in a statement are evaluated by the order of precedence

```
unary > binary > keyword
```

and left to right.

```
1 + 2 * 3 " equals 9, because it evaluates left to right"
1 + (2 * 3) " but you can use parenthesis"

1 to:   #(a b c d) size   by:   5 - 4
 "is the same as:"
1 to: ( #(a b c d) size ) by: ( 5 - 4 )
```

If you want to send many messages to the same object, you can use the cascading operator `;` (semicolon):

```
OrderedCollection new
  add: #abc;
  add: #def;
  add: #ghi;
  yourself.
```

"This first sends the message #new to the class OrderedCollection (#new is just a message, not an operator). It results in a new OrderedCollection. It then sends the new collection three times the message #add (with different arguments), and the message yourself."